Using Perl for Bioinformatics
Module 1: Learning Perl

Jason Stajich, jason.stajich@duke.edu

July 31, 2001

Contents

1 Introduction and FAQ

1.1 Course Scope

This one day course will attempt to teach you the basics of programming in Perl including how to write simple scripts, its applications to some common problems in life science research, and pointers to sources of information to assist you in the future.

1.2 This document

This document was prepared using the LaTeX typesetting system. All examples cited in this document are available in the directory where the document source is found.

This document is intended to be a short summary of the perl language, but will not be as detailed as other books or the perl built-in documentation about the perl language.

1.3 Why use perl for bioinformatics ?

Perl has seen widespread use in bioinformatics because of its ease of use, extensiblity, and its ability to scale to complex tasks. Being a scripting language, new programmers do not have to wrestle with the complexity of a compiled language and can quickly write scripts to achieve the small - medium sized tasks that they need to accomplish quickly. Perl was built to provide easy string manipulation and has simple I/O mechanisms making it ideal for formatting, searching, and reorganizing data sets contained within files. The **P**ractical **E**xtraction and **R**eporting **L**anguage was built for simplifying system administration tasks like *How many people logged in over the last week on Machine X?* The problem space of much of bioinformatics is data manipulation, text processing, and simple summary calculation which perl is perfectly suited to handle.

1.4 Help I'm stuck and I can't debug

When learning a new environment, be it programming a new language or performing a new lab protocol, it is helpful to know where to search for answers. Perl has a built-in documentation system that can get you started. Additionally web resources and books have been published which provide avenues of help to FAQ and difficult questions alike. For questions of a biological nature, online forums such at newsgroups bionet.software and bionet.biology.computational are useful as well as the Bioperl project's mailing list. Pointers to these web resources are located in *Resources* section.

The built-in perl documentation system can be accessed by using the command perldoc. It is analagous to the man system in UNIX. Typing in perldoc and a Module name or -f Function Name will display the manpage like documentation of the module or function name.

```
% perldoc perl
  show a listing of available documentation modules about perl.
% perldoc perlsyn
  describe the syntax of the perl language.
% perldoc perldoc
  This will show the documentation for the perldoc system
% perldoc -f split
  This will show the documentation on the perl function 'split'
% perldoc File::Spec
  This will show documentation for the File::Spec module if installed
  on your system.
```

Spelling and case do matter. If you aren't sure about the spelling of a module's name one can use the -i flag to specify searching - for example the following will display all the modules who start with File

```
% perldoc -i File
```

Additionally, there is online help for all CPAN modules. Go to `http://www.cpan.org` and follow the directions for searching for a particular module.

2 Perl Basics

Perl is an interpreted language and so all code that one writes runs through the perl interpreter called 'perl' on most systems. So one writes perl code and runs it through the interepreter. This tutorial will familiarize you with the perl language and its syntax.

One of the first mantras you must come to understand about perl is *"There's more than one way to do it" (TMTOWTDI)*. This has many ramifications when trying to teach a perl class because often one might only be taught how to do something one way, when in fact there are a number of other ways to achieve the same outcome.

The first example of this has nothing to do with the perl language, but is in the execution of scripts that are written in perl code. One can either specify the location of the perl interpreter in the format `#!/usr/local/bin/perl` OR one can omit this and run the script with the command `perl scriptname.pl`

On a typical UNIX system the perl interpreter is installed in the directory ``/usr/local/bin/perl''. If we were in MS Windows system the path to the perl interpreter might be "`C:\Perl\Perl.exe`".

Additionally when specifying the name of the interpreter one can also pass in command line arguments. For example the *-w* flag is used to have perl warn the user about all warnings instead of just suppressing non-fatal ones.

2.1 Hello World

```
#!/usr/local/bin/perl -w
perl ``Hello World '';
```

Figure 2.1: HelloWorld.pl

The code in Figure 2.1 shows a a simple example of how to print out the message "Hello World". Line 1 is the specification of where the perl interpreter is located. Line 2 is the command to print out some text, in this case the string "Hello World" followed by a newline.

2.2 Variables, Literals, Strings and Quotes

Perl has support for 4 types of variables.

- Scalars - $scalar_variable
- Arrays - @array_variable
- Associate Arrays (Hashes) - %hash_variable
- GLOBS (Data streams) - *STDOUT

Perl supports the literals: chars, integers, floating points, references, and strings all within the scalar type. Perl then casts a scalar to an appropriate type based on context. Example 2 in Figure 2.2 shows how a variable containing a number can be treated in either a numeric or character context.

```
$a = 100;
print ``Give me a $a dollar bill.\n''
print ``Twenty times 10 = '', $a * 20, ``.\n'';

> perl example2.pl
Give me a 100 dollar bill.
Twenty times 10 = 2000.
```

Figure 2.2: example2.pl and output

Numbers can be declared as a string or a literal number. The following are all equivalent.

- `$num = 100;`
- `$num = ‘‘100’’;`
- `$num = 1e2;`

Floating point numbers can be declared in the same way all other numerics are declared.

```
$double = 1.0023;
$float  = '1.7890102';
$irrational = 1/3;
```

By putting the numeric in a string, it is stored as a string and only casted to a particular numeric depending on the context it is used.

Characters are equivalent to strings of length 1 and so there is no need to make a distinction between characters and strings as there is in languages such as C or java.

In fact, the ease of using and manipulating strings in perl are one of the strongest attraction to the language especially for creating quick and dirty scripts to transform data.

Strings can be declared in a number of ways using different quoting operators and functions. Perl makes a distinction between the single quote ' and the double quote “. The single quote is used to declare a string as literally what is contained within the quotes without outside references. The double quote is used to declare a string but process any references to other variables and use them to build the string. These are described in Example 3 found in Figure 2.2.

Additionally strings can be built as composites of other strings in much the same way the double quote operator does. The '.' operator takes 2 strings and returns a new string that is the concatenation of the two. This is equivalent to the '+' operator in java. The sprintf function may be familiar to C programmers as it allows one to produce a formatted string using the universal C conventions. More information can be obtained from the perldoc system by issuing perldoc -f sprintf.

The backtick ‘ is a different beast, it is used to execute programs in the local shell as in `$datetime = ‘date‘;`. Executing external commands and programs will be covered in the Advanced section.

```
# initialize the variable $a with the string literal 'yellow'
$a = 'yellow';

# initialize the $b to be a new string of $a concatenated with " submarine"
$b = "$a submarine";

# print out the value of $b and a newline (a return)

# print accepts a list of things to print separated by commas
# this is because print accepts a list to print - thus it is
# accepting a list of items to print, and everything can be
# converted to a string

print $b, "\n";

# help show the difference between single quote (')
# and double quote (")
$c = 'green is the opposite of $a';

print $c, " is the single quoted message\n";

# concatenate 3 strings together
```

```
print "Curious George and the man with the ". $a . " hat" . "\n";

print "---------\n";
print "make sure you use double quotes when printing \nspecial ",
'characters like \'\n\' and \'\t\' '.
'or else they show up literally', "\n\n";

print "if you want to print a double quote \neither escape it \"",
    ' or use single \' to print it - "', "\n";
```

Figure 2.2: example3.pl

2.3 Arrays and Lists

Arrays and lists are ordered collections of items. The different between a list and an array is academic, an array is a variable that contains a list while a list is simply an anonymous listing of items. The first part of the example in Figure 2.3 might make it clearer. Arrays and lists do not need to contain all the same type of items, an array can contain numerics, scalars, or references. However, arrays cannot contain other arrays directly - perl always tries to flatten arrays of arrays into a single array. If one is interested in multi-dimensional arrays (arrays of arrays) it is necessary to use the notion of an array reference. This is declared with the [] operators around the list one wants to declaring as an array reference. Perl also uses the special operator \@array to indicate a reference to an already existing array. The operators [@array] and \@arrays are then equivalent for all intensive purposes.

Arrays can be indexed via the the [] operator. To get the 1st item in a list we use $array[0] (arrays start at 0 like in C). There are a number of functions for use with arrays. The pop and push methods treat a list like a stack by removing and adding elements to the end of the array. shift and unshift are their cousins operating on the front of the array instead. The splice function can operate anywhere in the array and can insert or remove elements anywhere in the array.

```
# a list - it is not stored in a variable so it is anonymous
(1, 200, '90.5', 'english');
# an array
@array = ('a', 'b', 'c');
print "1st array element is '", $array[0], "'\n";

# can mix elements in an array
@array = ('a', 1, '89', 2.34);

# some array functions

# build an array by taking things that are
# separated by whitespace
@array = qw(1 dog barked);

# elements of the array can be joined together
# with the 'join' command which will join 2
# elements together
print "elements are -- ", join(' ', @array), "\n";

# arrays are flattened in perl
@array = (100, 200, (220, 230, 280), 300);
```

```
print "element list is ", join(',', @array), "\n";

# array references allow us to build arrays of arrays
@matrix = ( [ qw( 67 78 98) ],
    [ qw( 99 95 82) ], );

print join("\t", qw(Test1 Test2 Test3)), "\n";

# will discuss loops and conditionals later on, but
# here is how to see each 'row' in the matrix and
# print the average score\n";
foreach my $row ( @matrix ) {
    # use @$row to dereference an array reference
    $total = 0;
    foreach my $col ( @$row ) {
$total += $col;
    }
    $average = $total / scalar @$row;
    print join("\t", @$row), "\t--> ",  $average, "\n";
}

# pop and push work on the end
push @array, 1;
my $one = pop @array;
die unless ($one == 1);

# shift and unshift work on the front
unshift @array, 'bee';
my $two = shift @array;
die unless ($two eq 'bee');

# splice can operate anywhere on the array
# remove the elements at position 1 and 2
print "array is ", join(',', @array), "\n";
my (@elements) = splice(@array, 1, 2);
print "array is ", join(',', @array), "\n";
# get out the elements at position 1 and 2
# and replace them with new elements
my (@elements) = splice(@array, 1, 2, (109, 207));
print "array is ", join(',', @array), "\n";
```

Figure 2.3: example4.pl

In perl everything can be treated like a list. This derives in part from the mantra *TMTOWTDI*. The fact that everything can be treated as a list makes argument passing very simple - one only has to expect a list of arguments and can process them based on what is expected. It also lets you be an extremely lazy programmer and since perl does not enforce strong typing of variables one can get into trouble. There are mechanisms for supporting strong typing in certain contexts, but it puts the burden of checking on the programmer.

2.4 Associative Arrays or Hashes

Associative arrays or hashes allow one to index a list of items on arbitrary keys rather than numerical indexes. They are equivalent to the concept of hashtables in Java or C++ and provide fast lookups. The examples in Figure 2.4 show how one can use an associative array to index things in different ways. To get or store an item on a specific key one uses the { } operators to do the lookup. If on wanted to store an array hashed to a specific key, we would operate in the same way as with arrays - by storing an array reference at the location.

```
# associative arrays can be declared in different ways
# declared and initialized at the same time
my %hashmonths = ( 'Jan' => 1, 'Feb' => 2, 'Mar' => 3);

# declared, then initialized
my %hashdays;
$hashdays{'Sun'} = 1;
$hashdays{'Mon'} = 2;
$hashdays{'Tue'} = 3;
$hashdays{'Wed'} = 4;

# a reference to a hash
my $hashref = { 'Jul' => 7, 'Aug' => 8};

# get all the keys in a hash
my @keys = keys %hashmonths;
print "keys are ", join(',', @keys), "\n";
# get all the values in a hash
my @values = values %hashdays;
print "values are ", join(',', @values), "\n";
# notice that insert order is not necessarily preserved

# iterate through data
while( my ($day,$val) = each %hashdays ) {
    print "$day = $val\n";
}

use Tie::IxHash;
tie %orderedhash, "Tie::IxHash";

$orderedhash{'Guitar'} = 'B.B.King';
$orderedhash{'Trumpet'} = 'Dizzy Gilespie';
$orderedhash{'Accordian'} = 'Weird Al';

foreach my $key ( keys %orderedhash ) {
    print "Instrument '$key' played by $orderedhash{$key}\n";
}

# store an array reference in a hash entry
$orderedhash{'Guitar'} = [ 'BB King', 'Eric Clapton'];

# add some more people
push @{$orderedhash{'Guitar'}}, 'Jimmy Page';

# store a hash ref in a hash
```

```
my %hash3 = ( 'canine' => {
  'domestic'  => [ 'hounds', 'labs', 'retrievers'],
  'wild'      => [ 'grey', 'siberian' ]},
      'feline' => {
  'domestic' => [ 'calico', 'tabby', 'siamese', 'persian' ],
  'wild'     => [ 'panther', 'lynx', 'bobcat', 'lion',
  'tiger' ]});

foreach my $animal ( keys %hash3 ) {
    print $animal, "\n";
    while( my ($domain_name,$domain) = each %{$hash3{$animal} } ) {
       print "\t$domain_name\n";
       foreach my $type ( @$domain ) { print "\t\t$type\n"; } }
}
```

Figure 2.4: example5.pl

2.5 A Brief Discussion of Truth

The examples in this document utilize the flexibility of perl with regard to truth without informing the reader. In perl, everything can be tested for whether it is true or false. The rule is that non-zero, non-empty values are considered true, everything else is false.

In the case of a loop such as a foreach loop (will be discussed in the next section) the loop will iterate through all the items in the list. In the case of a while loop, the loop will iterate until the condition is false - the list is empty.

```
if( @a ) { # true of @a is non empty}
if( %a ) { # true if %a is non empty}

while( @a ) { # must remove items from @a to prevent infinite loop}
```

2.6 Conditionals and Loops

Perl has the standard constructs for conditionals.

if..elsif..else

logically opposite unless..elsif..else

Loops are supported in perl with the following constructs.

for(initialize; conditional; iterate) { }

while(conditional)

do {} while(conditional) -

foreach $itemptr (@items) { }

2.7 Built-in perl functions

2.7.1 String functions

Since a large amount of data manipulation in perl centers around strings, there are quite a few built in functions for operating on strings.

- *split* splits a string into an array based on a separator. It takes as arguments, the separator pattern, the string, and optionally the number of items to split the string into and will return an array of items. If no number is entered for this number then *split* will attempt to break up the string where all separators exist. Otherwise it will only break up the string into the first N-1 items and store the rest of the string in the Nth item. The separator value can be a regular expression (see Section on Regular Expressions) a string. So one can pass in `/\s+/` to match any whitespace or ':' to match semi-colons as separators in a list.

- *join* joins a list of items into a single string separated by a specified string. It takes as arguments the separator string and a list of items to join together and returns a single string. It is a convient way of viewing and manipulating data when one only wants to perform an operation between all consecutive pairs in a list - the "fencepost" problem.

- *substr* returns a substring of a specified string. It takes as arguments an input string, the offset point in the string (strings start at 0 like arrays), and optionally the length of the substring to extract. If no length is specified then the substring returns everything from offset to the end of the string. The offset can be negative to indicate starting at the end of the string. substr also takes an optional 4th argument which specifies a replacement string to replace the substring with in the query in an analagous way to the splice function on arrays.

- *sprintf* and *printf* can be used to create and print formatted strings. They both use the standard C syntax of a formatting string and then a list of items to use in the formatting. *sprintf* returns a string while *printf* takes an optional 1st argument which specifies the filehandle to print the string to (STDOUT is implied when no filehandle is specified).

- *print* is a simplified form of *printf*. It accepts an optional filehandle and a list of items to print to the filehandle. If no items are specfied *print* will print the implicit `$_` variable (see the Section on Special Variables)

- *chomp* and *chop* are used to remove trailing input record (in most cases this is "`\n`"). It is most useful when reading in data from an input loop and the newline separator is not already removed. *chop* is the unsafe version of chomp in that it will delete the last character of a string regardless of whether or not it is the record separator. chop and chomp will also operate on a list by processing each item in the list.

- *lc* and *uc* respectively will return a lowercase and uppercase version of an input string. *lcfirst* and *ucfirst* only lower and uppercase the first letter in a string.

- *index* and *rindex* will find the first occurance of a substring within a specified string. They take as arguments the query string, the sub string, and optionally the offset to start from. *index* starts at the beginning of the string while *rindex* starts searching from the end.

- *reverse* will return a copy of the input string exactly reversed.

- `q{STRING STRING}` will return a string literally as what is typed between the { } (which can be subsituted for by any other pairing of separator ((), `/ /`, | |).

- *chr* and *ord* return the character and ascii value respectively for an input number or character. *chr* will return the character value for an ascii value and *ord* will return the appropriate ascii value for a character.

2.7.2 Array functions

- *scalar* returns the number of items in an array. The special operator `$#` returns the index of the last item in the array, effectively scalar @array - 1;

- *map* evaluates an operation block and operates on each item in a list. It takes as arguments a BLOCK and list of items to operate on and returns a list of values for each of the operations. A good example is @chars = map(chr, @nums) which will return a list of characters for a given set of ascii values.
- *grep* will evaluate a given block or regexp on all items in list and return the items which evaluated to true. It behaves much in the same way UNIX grep does except that it is operating on a list of items rather than a file.
- *reverse* behaves exactly as reverse on a string works returning an entire array reversed.
- *sort* returns a list sorted based on the input sort function and the array. If no function is passed the default alphanumeric least to greatest is applied.
- *splice*, *push*, *pop*, *shift*, *unshift* were described earlier and only operate on a real array of data not an anonymous list.

2.8 Input/Output

Perl I/O is by default incredibly easy and can be tweaked in sophisticated ways to handle non-default behavior. The main operators and functions involved in Perl I/O are the *open* function and <> operator.

The *open* function is used to open filehandles. The syntax is to specify the name of the filehandle and the filename or system operation on wishes to perform. The examples in Example 6 in Figure 2.8 show how this can be done.

For more advanced users one can either open handle to a program which will be producing output or accepting input. To manipulate a program which will be producing and receiving output one must use the module IPC::Open3 which requires a fairly sophisticated understanding of how perl IO works.

```
open(IN, "< example6.pl"); # read this script
# and print it to STDOUT...
while(<IN>) {
    print;
}
close(IN);
print "-----\n";
open(PERLDOC, "perldoc -f open |");
# get the perldoc page on the open function
while(<PERLDOC>) {
    print;
}
close PERLDOC;
print "-----\n";
open(WC, "| wc");
print WC "a simple mesage\nwith 2 lines 10 words 52 characters\n";
close WC;
```

Figure 2.8: example6.pl

2.9 Regular Expressions

Regular expressions are the bread and butter of Perl. The popularity of the perl regexp syntax and ease of use has spawned the GNU project to create a perl regexp library to allow easy porting of regexps to different language environments. We will walk through a few examples in this text and the in-class tutorial but I commend the reader to some of the numerous web resources and books that are available for perl.

Regular expressions are usually invoked within the perl pairing `/ /`. There are a few special characters in regular expressions which have special meaning.

- `\s` - match whitespace (any tab, space, newline)
- `\S` - match anything EXCEPT whitespace
- `\d` - match any digit (0-9)
- `\D` - match anything EXCEPT a digit
- `\w` - match a word (any alphanumeric)
- `\W` - match anything EXCEPT an alphanumeric

Additionally quantifier operators are defined

- `*` - match expression 0 or more times
- `+` - match expression 1 or more times
- `?` - match expression 0 or 1 times
- `{n}` - match exactly n times
- `{n,}` - match at least n times
- `{n,m}` - match at least n but not more than m times

Within the regular expression the [item] operators allow the programmer to specify a list of items that are allowed to match, providing some flexiblity for matching multiple patterns.

Perl defines some assertion operators:

- `^` - assert that match begins at start of string
- $ - assert that match ends at end of string

Pattern matching can be initiated with a few different operators. Enclosed by the `/ /`, a pattern can be done on the implict variable `$_` or by using the `=~` (positive assertion) or `!~` (negative assertion) operators. Figure 2.9 demonstrates these.

When patterns are enclosed in parentheses in a regular expression the regular expressions stores the match. If it is the first set of parentheses, the data is stored in the variable $1, if it is the second set then $2, and so on. One can also write a one-liner (one-liners are a Perl staple) to match and store the values in parens. See Figure 2.9 for an example.

```
if( /^(\d+)/ ) { # match leading numerics in $_
 print ‘‘$1 is the match\n’’;
}
# match trailing numerics w/ or w/o whitespace
# in variable $line and store in variable $num
my ($num) = ( $line =~ /(\d+)\s*$/ );

if( $str !~ /\S+/ ) { # test if str does not contain any non-whitespace
}
```

Figure 2.9: One-Liners for pattern matching

So let's write a regular expression which matches the zip code from a list of addresses. See Figure 2.9 for the code.

```
my @data = ( '1 Infinite Loop, Cuperteno, CA, 95014-2084',
     '201 Vassar St, W59-200, Cambridge, MA 02139');

foreach my $address ( @data ) {
    chomp;
    if( $address =~ /(\d{5}(-\d{4})?)$/ ) {
        print "zip is $1\n";
    }
}
```

Figure 2.9: example7.pl

2.10 Scope, "use strict"

In some of the examples above the reader may have noticed the directive 'my' used when initializing a variable. This directive is to declare a scope for a variable. This scope is determined by curly braces ({ }). So typically the scope in our examples is for the entire script but in some cases like the following:

```
foreach my $v ( @list ) {}
```

The scope of $v is only for the duration of the foreach loop, each time the loop iterates a new instance of the $v variable is created. If one tried to access the value of the variable after the loop a null would be returned.

Perl is a language which allows a programmer to be very lazy. By default all variables have the scope of the entire program. This can be useful for the quick hack, but disasterous for the programmer of a large program attempting to debug a problem. Perl does provide a way to make the programmer more careful. By including the module 'strict' by simply adding `use strict;` at the beginning of your program Perl will force the programmer to declare the scope of variable before it can be used. This is a very good habit to get into when writing perl programs and will prevent a large number of subtle bugs from being written.

2.11 Subroutines

Subroutines are reusuable bits of code lumped together with a defined calling signature and should be familiar to those with programming experience in most any language. The word subroutine, function, and method are used interchangeable in this document however they do mean slightly different things to some people.

Because perl does not have strong typing subroutines are expected to accept 0 or more arguments in a list and return a (possibly empty) list. This allows for very flexible definitions and puts the burden on the programmer to make sure their method is called correctly. There are ways to assert the proper number of arguments are passed in with the correct type, but these are all ultimately the responsibility of the programmer to implement and not defined in a function signature as in java or C.

Subroutines can be declared anywhere in a perl script, the do not need to occur before their invokation in a script because the perl interpreter reads in the entire perl script, compiles it, and executes it in the interpreter rather than running each command as it is read. Subroutines are declared by simply specifying `sub subroutinename` and a set of braces ({ }) containing the code to execute. The implicit variable `@_` provides access to the arguments passed in to a method. Example 8 in Figure 2.11 shows how to declare and use a subroutine. The leading & is not necessary when calling the routine 'jump' but it helps someone reading the code discern that 'jump' is a subroutine and not a built-in perl function.

```
# a random walk in 1 dimension

sub jump {
    my ($start, $size) = @_;
    my $sign = rand() > .5 ? -1 : 1;
```

```
    return $start + ( $size * $sign);
}
my $start = 0;
print "started at $start ...";
foreach ( 1..1e5) { # iterate 10,000 times
    $start = &jump($start,1);
}
print " ended at $start\n";
```

Figure 2.11: example8.pl

3 Advanced Perl

3.1 References

References in perl are a location in memory where the variable is stored or in C terms, a pointer to the object. References are very useful for handling multi-dimensional arrays, storing arrays or hashes within hashes, and for manipulating data in-place within a subroutine or loop without having to copy the original and changed value back and forth.

The typical way to make a reference to any value in perl is to prefix it with a \. Figure 3.1 shows a number of different ways to interact with references. Typically one can use a reference to manipulate data in-place such as passing an array reference to a function and having the data updated.

```
my @array = reverse(1..30); # 30 -> 1
my %hash  = ( 'Jan' => 1,
      'Feb' => 2,
      'Mar' => 3);
my $scalar = '1900';

print "array ref is ", \@array, "\n";
print "hash ref is ", \%hash, "\n";
print "scalar ref is ", \$scalar, "\n";

my $arrayref = \@array;
my $hashref  = \%hash;
my $scalarref = \$scalar;

print "scalar value is ", $$scalarref, "\n";
print "array index 1 is ", $arrayref->[1], " index 2 is ", @$array[2], "\n";
print "hash has ", scalar keys %{$hashref},
    " keys and Feb value is ", $hashref->{'Feb'}, "\n";

my @one = (1,2,3);
my @two = (4,5,6);
foo(@one);
print "one is @one\n";

foo(\@two);
print "two is @two\n";
sub foo {
    my ($item) = @_;
    my $a;
    if( ref($item) =~ /array/i ) {
```

```
        $a = $item;
    } else { $a = [ @_ ] ; }

    foreach my $v ( @$a ) {
        $v += 10;
    }
}
```

Figure 3.1: example9.pl

3.2 Special Variables

Perl has a number things that make it cryptic to novice users. While this may seem like a secret club, the special variables that are used in perl are relatively straight-forward once you have a cheat sheet.

- `$_` - implict variable assigned during loops
- `@_` - implict ARG variable in subroutines
- `$/` - input record separator variable (defaults to "`\n`")
- `$^O` - Operating System name is stored here
- `$$` - Process ID
- `$@` - eval error is stored here
- `$,` - output field separator

3.3 Executing external programs

In typical Perl fashion there are a number of different ways to execute remote programs from within a perl script. All the methods open a new shell on the system and execute the programs with uid of the perl script and thus have the same path as the uid running the script. This can cause problems for developers who debug CGI scripts with their own UID and then copy them to the webserver. Sometimes a developer will have neglected to refer to the full path for a program or a webserver uid does not have the same access rights as the developer. The CGI program will then fail at these instances of executing a remote program.

The most simple is with backticks as in the following.

```
`/bin/cp $file1 $file2`
```

However output is sent to STDOUT and not sent to the program. The `qx//` operator is equivalent.

```
qx/cp $file1 $file2/
```

The output from the backtick or qx operators can be assigned to a variable. See first example in Figure 3.3. It is necessary to have the *chomp* operator in the examples because the date function returns a line with a newline.

The *system* command executes a command and only returns the return code for the application rather than the output of the program. *system* takes as input an array and expects each command line option to be a separate entry in the array.

A third way to execute programs and getting their output is by handing them to the *open* command and executing the program within this command. This opens a shell on the system and runs the command.

There are other ways to execute programs using the fork command and the IPC::Open3 to read and write I/O for a program within perl.

```
my $date;

# get the date with backticks
chomp($date = `date`);
print "date is '$date'\n";

# get the date with qx cmd
chomp($date = qx/date/);
print "date is '$date'\n";

# get the date with the system command
$date = system("date");
print "date is '$date'\n";

open(DATE, "date |");

chomp($date = <DATE>);
print "date is '$date'\n";
```

Figure 3.3: execution.pl

3.4 Object Oriented Perl

While Perl does not explictly force object orientation, it does have mechanisms for introspection, inheritance, polymorhphism, and interface defining albeit crude at times, it allows users to build toolkits and applications with object oriented principals and expect users of these toolkits to follow a protocol. The following list describes some of the mechanisms for OO programming in perl and Example 10 in Figure 3.4 shows a very simple package for ticket sales.

- The *package* directive allows the definition of a set of methods to be associated with a particular class name. This allows one to build classes that interact with one another without having just a jumble of sub routines lying around.
- The *bless* directive declares an object to be of a particular type. Typically one *bless*es a hash to be particular package and then it has access to all the methods of that package.
- By defining a class variable @ISA inheritance is supported. One can have multiple inheritance, however if two parent classes have the same function only the method from the first class listed in the @ISA array will be derived to the child.
- The *ref* method allows one to inspect what type of object on has. *ref* will return null when operating on a scalar and will return the object reference value when inspecting a reference, when inspecting a package blessed object it will return the name of the package. The associated method *isa* will check the @ISA array (and all its parental dependancies) for a blessed object and see if the queried name of a packages matches any of those used to derive the object.

Much more information is available about OO perl in Damian Conway's book *Object Oriented Perl*

```
# code to use the package

my $seller = new TicketSeller(10);
my $total = 0;
while ( $seller->buy_ticket ) {
    print "Bought a Ticket\n";
    $total++;
```

```
}
print "$total tickets sold\n";

# package defined
package TicketSeller;

use strict;

sub new {
    my ($class, @args) = @_;
    my $self = {
       'numtickets' => shift @args
    };
    bless $self, $class;
    return $self;
}


sub buy_ticket {
    my ($self) = @_;
    if ($self->{'numtickets'} > 0) {
         $self->{'numtickets'}--;
         return 1;
    }
    return 0;
}

1;
```

Figure 3.4: example10.pl

4 Resources

4.1 Books

There are a number of good perl books out there. I will list a few of the ones that are good starters. The O'Reilly series is usually good, but may not provide enough detail for novice programmers. Damian Conway's book is not for the faint of heart

- *Learning Perl*, Randal Schwartz and Tom Phoenix. O'Reilly & Associates. 3rd edition July 2001.
- *Programming Perl*, Larry Wall, Tom Christiansen, & Jon Orwat. O'Reilly & Associates. 3rd edition May 2000.
- *Advanced Perl Programming*, Sriram Srinvasan. O'Reilly & Associates. 1997.
- *Object Oriented Perl*, Damian Conway. Manning Publications. 1999.

4.2 WWW

A number of great online tools exist for learning perl and programming for bioinformatics.

- `http://stein.cshl.org/genome_informatics/` - Lincoln Stein's CSHL Genome Informatics course has many of the lectures and references linked online.
- `http://www.perl.com` - O'Reilly's everything perl site. Some reasonable tutorial links are there
- `http://bio.perl.org` - Bioperl site has links about bioinformatics and perl.
- `http://www.comp.leeds.ac.uk/Perl/start.html` Nik Silver's page - very basic, but much like this document.
- `http://www.netcat.co.uk/rob/perl/win32perltut.html` Robert Pepper's Perl Tutorial including some Win32 specifics.